Defensive Programming

Writing Better Code, Reading Code Better

Oleg Goldshmidt

olegg@il.ibm.com

Haifa Research Lab

Agenda

- What is good code and why is it important?
- How to avoid a lot of frustration and make our work a lot more interesting?
  - A few words about design, prototype, comments, documentation, error handling, testing, and all that jazz...
- What is “defensive programming”?
- Some useful tools and techniques...

Writing Better Code

- requirements
- specs
- prototyping
- design
- reviews
- error handling
- logging
- version control
- testing (including regression testing)
- defensive programming
- WHY??? How does it make code better???

Saturday Night Fever

- Saturday (or Friday) night: you are getting ready for
  - a hot date
  - your mother-in-law's birthday party
- your boss calls:
  - an irate client thinks there is a problem, threatens to switch to competitor
  - the perceived problem is in John Smith's code
  - John is vacationing in Nepal without a cell phone
- your task:
  - is there a problem? in John's code? what is it?
  - fix, test, stage, roll to production
- time estimate: 15 minutes, 3 hours, or a week?

A More IBMish Scenario

- a customer complains of a problem in last year's code
- you are working on a different project, an important deadline is the day after tomorrow, you are way behind schedule
- John Smith who wrote that file last year is trekking in Nepal without a cell phone
- IBM-specific options:
  - ignore the client
  - demand $$$$ for the fix
  - delay the current project
- what are you willing to do?

How Do You Spend Your Time?

- in a typical SW development project — in the debugger
  - unless you spend time on requirements, prototyping, design, reviews, error handling, logging, testing, etc.
  - is it a fair tradeoff?
- debugging is the worst possible way to waste time
  - requires immense patience and concentration
  - often very difficult to do (e.g., concurrency)
  - most people find it uninteresting and hate it
  - unproductive:
    "I'm afraid that I've seen too many people fix bugs by looking at debugger output, and that almost inevitably leads to fixing the symptoms rather than the underlying problems." [Linus Torvalds]

Avoiding the Debugger

- ideal scenario: a customer complains of a problem
  - verify there is a problem
  - where is the problem?
  - what is the (root cause of the) problem?
  - design, implement a fix
  - test: does it fix the problem? does it break anything else?
- how to reach this blissful state?
  - infrastructure and instrumentation:
    - design, review, error handling, logging, testing
  - keep in mind:
    - you will not maintain the code you write
    - you will maintain code that others wrote

How to Write Better Code?

- design
- prototypes
- comments and documentation
- review
- errors and logs
- testing
- defensive programming
- tools

Design, Prototypes, and Reviews

- “The sooner you start to code, the longer the program will take.” [Roy Carlson]
- “If you can’t write it down in English, you can’t code it.” [Peter Halpern]
- “If you have too many special cases, you are doing it wrong.” [Craig Zerouni]
- “Get your data structures correct first, and the rest of the program will write itself.” [David Jones]
- “The structure of a system reflects the structure of the organization that built it.” [Richard E. Fairley]
- “Prototyping cuts the work to produce a system by 40%.” [Larry Bernstein]

Writing Code for People

- "Programs must be written for people to read, and only incidentally for machines to execute." [H. Abelson, J. Sussman]
- comments
  - "If the code and the comments disagree, then both are probably wrong." [Norm Shryer]
  - comment extensively when you do something non-trivial, e.g., optimize by hand
- documentation
  - "Don't include a sentence in documentation if its negation is obviously false." [Bob Martin]
- explain why, not what — what is clear from the code
- there will be changes — think in the future tense!

Errors and Logs

- fail early!
  - (sometimes presented as an IBM slogan, though I doubt it)
  - compile-time, link-time, test-time, and run-time errors
- return useful error codes and messages
  - garbage in — useful error message out
    - not just for user input: if a function assumes something about its input — check it
    - `assert(3)` is usually not the right tool
- provide an extensive logging facility to track the progress of the code — much easier and more useful than stepping through in a debugger
- provide a warning facility (more of this later)

Testing

- “It takes three times the effort to find and fix bugs in system test than when done by the developer. It takes ten times the effort to find and fix bugs in the field than when done in system test. Therefore, insist on unit tests by the developer.” [Larry Bernstein]
- “Testing can show the presence of bugs, not their absence.” [Edsger W. Dijkstra]
- We want BUG-FREE code! Testing is essential, but not enough!

Example from Real IBM Code

- `#include "foo.h"` or `#include <foo.h>` ?
- `#include <ldap.h>` — BUG!!!
- first test on a computer with OpenLDAP installed:
  - declaration:

    ```
    int Foo::bar(int sth, LDAPCtrl** ctrl=0);
    ```
  - call: `Foo foo; foo->bar(5);`
  - OpenLDAP:

    ```
    typedef struct ldapctrl {...} LDAPCtrl;
    ```
  - custom LDAP:

    ```
    typedef struct _LDAPCtrl {...} LDAPCtrl;
    ```
  - the linker can't find

    ```
    int Foo::bar(int,ldapctrl**)
    ```

No Trivial Bugs

- most bugs are symptoms of a design problem
  - you will not figure it out stepping in a debugger
  - must pay attention even to simple things
- trivial example: shadowing (real IBM code)

```
class Object {
    public:
        long shadow;
        static void Cast(long shadow);
};

void Object::Cast(long shadow)
{
        shadow = 1L;
}
```

Paying Attention

- what do I do when I write/read code?
- design issues require understanding
- but what about the “symptoms”?
  - practically impossible to notice by naked eye
  - there are (relatively simple) rules, but applying them is too tedious
  - computers are much better than humans applying rules automatically
- code analysis tools

Know Thy Compiler

- modern compilers are able to warn you about many different symptoms of deeper problems
- you won't win the lottery without buying a ticket
- ask the compiler to warn you, and it will
- do not ignore the warnings!
  - do not treat them like a nuisance
  - investigate in depth: is there a design problem the compiler tries to warn you about
    - the compiler won't tell you: it has no understanding of your design intent

Portability

- platform: processor architecture, OS, libraries, compilers, linkers
  - assume just 3 of each: $3^5 = 243$ platforms easily
- practically impossible to test all combinations
  - do you have access to all?
  - do you have enough HW?
  - do you have enough time?
  - when you employer or client decides to buy another type of machine, will your code still work?
- yes, it is possible to develop portable code
- Who cares? My code only needs to run on Windows.
  - war story: port to Linux in 2 months or you are fired!
  - what about the next version of MSVS / `cl.exe` ?

Portability II

- standards: POSIX (and Windows), ANSI, XOPEN, etc.
- compilers are supposed to implement language standards, but normally only approximate them
  - some are better than others
  - may implement many extensions, but warn about non-standard constructs
  - choose a compiler that is portable and compliant for development
    - but don't forget to test on important target platforms
- same goes for linkers, although they are more esoteric
- treat build system is a part of the code!
  - and make it portable

Thinking in the Future Tense

- real IBM header file:

```
// [name withheld to protect the guilty - OG]
// Apr-09-2003
// I removed the non-Win32 leg since the
// pthread_cond_wait code did not always work
// (needed to be waiting when signal set or
// signal was lost). Since all platforms now
// either are Win32 or have Win32 emulation
// layer, just use the Windows API's
```

- guess what: the code had to be ported to a platform without a Win32 emulation layer, and there was no version control...

GCC Examples

- very portable compiler
- implements many extensions, but can be made very strict
- compile new code with `-DPOSIX_SOURCE`, `-D_ISOC99_SOURCE` , etc.
  - RTFM, RTF `/usr/include/features.h`
- use `-pedantic` for strict conformance
  - `-ansi` , `-traditional` for old code
- what about Visual C++? Use MSDN:

```
http://msdn2.microsoft.com/en-us/library/\
      9s7c9wdw(vs.80).aspx
```

Selected GCC Warnings

```
-Wchar-subscripts:
  Warn if an array subscript has type
  'char' (may be signed)
-Wformat
  Check calls to 'printf' and 'scanf',
  etc., to make sure that the arguments
  supplied have types appropriate to
  the format string specified
-Wunused
  Warn whenever a static function,
  argument, label, local variable,
  or computer result is declared
  but not used.
-Wfloat-equal
  Do not compare FP numbers for equality!
```

Selected GCC Warnings II

-Wparentheses

Do you remember the operator precedence table?

What is x<=y<=z? Are you sure?

Is this what the programmer meant?

```
{
  if (a)
    if (b)
      foo ();
  else
    bar ();
}
```

Falling Through `switch`

```
enum {
  FIRST_CASE = 1,
  SECOND_CASE,
  THIRD_CASE
} c;
int n;
...
switch (c) {
case FIRST_CASE: n = 1;
case SECOND_CASE: n = 2;
}
do_something_with(n);
```

To catch this, use `-Wswitch` , `-Wswitch-default` , or `-Wswitch-enum` .

Selected GCC Warnings III

```
-Wsequence-point
  C standard defines partial ordering in terms
  of "sequence points". Examples: after
  evaluation of an expression, after first
  operand of a &&, ||, ?:, or 'comma', before
  function call (but after evaluation of the
  arguments), etc. Only a partial order (no
  order between function arguments, between
  function calls, etc.)

  Can catch undefined behavior:
  a = a++;
  a[n] = b[n++];
  a[i++] = i;
  x = func(a[n],b[n++]);
```

Selected GCC Warnings IV

`-Wuninitialized`

- Sounds obvious? Have you thought about clobbering a variable with `setjmp` ?
- Need to understand the compiler: works only with `-O`
- Related: `-Winit-self`
- Works for structures, does not work for volatiles
- False positives:

```
switch (y) {
  case 1: x=1; break;
  case 2: x=4; break;
  case 3: x=5;
}
foo(x);
```

```
int save_y;
if (change_y)
  save_y=y, y=new_y;
...
if (change_y) y=save_y;
```

Lints

- originally a specific tool, now any static analysis tool that flags suspicious or non-portable constructs
- PC-Lint and FlexeLint from `http://www.gimpel.com`
- many different open source implementations
- specialized lints
  - C++ — `cpplint` : `http://sourceforge.net/projects/cpplint/`
  - security — `splint` : `http://sourceforge.net/projects/splint/`
- PC-Lint output analyzer: `http://sourceforge.net/projects/aloa-lint/`
- Eclipse plugin: `http://sourceforge.net/projects/eclint/`

DIY Lint

```
g++ -c \
  -O2 \
  -pedantic \
  -ansi \
  -Wall \
  -Wformat=2 \
  -Wmissing-include-dirs \
  -Wswitch-enum \
  -Wswitch-default \
  -Wunused-parameter \
  -Wextra \
  -Wfloat-equal \
  -Wno-endif-labels \
  -Wshadow \
  -Wpointer-arith \
  -Wcast-qual \
  -Wcast-align \
  -Wconversion \
  -Waggregate-return \
  -Wpacked \
  -Wpadded \
  -Wlong-long \
  -Wvariadic-macros \
  -DPOSIX_SOURCE=1 \
  -DISOC99_SOURCE=1 \
  -D_REENTRANT \
  -Dlint \
  -D__NO_STRING_INLINE
  "$@" -o /dev/null
```

The Most Important GCC Option

```
-Werror
  Convert all warnings into errors.
```

Further Tools

- Insure++ `http://www.parasoft.com`
  - an instrumenting compiler that catches lots of errors, including memory leaks and corruptions at compile time and at runtime.
  - use it as yet another compiler — always a good idea
  - it is really good at catching errors
- free memory debugging tools
  - Valgrind
    `http://valgrind.org`
  - ElectricFence
    `http://directory.fsf.org/ElectricFence.htm`

Summary

- Do the interesting stuff and let the computer handle the annoying details
- Use the tools intelligently — they are often much more useful than you think.
- Pay attention to what your tools tell you — they are not there just to annoy you
- They are not as intelligent as you are — investigate the root cause of each problem flagged, usually it is a design issue
- Think of the maintainer — it may be a good friend of yours or it may be you
- Think in the future tense!